Reified Types

Ralph E. Johnson
johnson@cs.uiuc.edu

Abstract:
Sometimes the built-in type system of a language is too inflexible or does not provide the features you need. You can build your own type system by creating objects that represent types, letting each application object know its type, and checking these types at run-time. Reified types can be an entrance into a world of reflective programming.

Also Known as:
Instance/Type, Item Description

Motivation:
Consider a factory with many different machine tools manufacturing many different products. Every order has to specify the kinds of products it requires. Each kind of product has a list of parts and a list of the kinds of machine tools needed to make it. One approach is to make a class hierarchy of the kinds of machine tools and the kinds of products. But this means that adding a new kind of machine tool or product requires programming, since you have to define a new class. Moreover, the main difference between different products is how they are made. If you specify how a product is made by the sequence of machine tools that is needed to make it, then a new kind of product won't need any new code, but can be specified by giving the sequence of machine tool types.

It is better to make objects that represent "kind of product" and "kind of machine tool". They are both examples of *reified types*[1], i.e. objects that represent types. Each particular machine tool knows what kind of machine tool it is, and each order knows what kind of product it is going to produce. Thus, there will be classes MachineTool, MachineToolType, Product, and ProductType. A ProductType has a "manufacturing plan" which is a sequence of MachineToolTypes. But a particular instance of Product was made on a particular set of MachineTools. This is important for detecting when a machine tool needs to be repaired.

If the design has MachineToolTypes and ProductType then it will be possible to define new ProductTypes without programming. This is good, because product managers, who are usually the ones who need to specify a new product type, usually aren't programmers. It will be possible to design a tools that product managers can use to define a new product type by specifying the manufacturing plan, defining the labor and raw materials needed, the price of the final product, and so on. As long as a new ProductType can be defined with subclassing ProductType, it will be possible for product managers to do their work without depending on programmers.

There are constraints between types. For example, the sequence of actual MachineTools that manufactured a Product must match the MachineToolTypes in the manufacturing plan of its ProductType. This is a form of type-checking, but it can only be done at run-time. It might not be necessary to check that the types match when the sequence of MachineTools is assigned to a Product, because this sequence will be built by iterating over a manufacturing plan to find the available MachineTools. However, scheduling can be complex, and errors

---

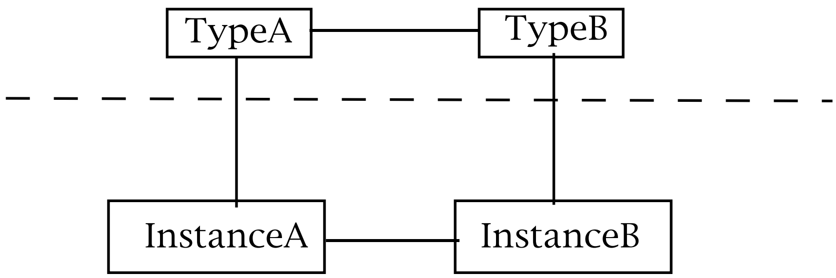
[1] "Reify" means "to regard something abstract as a material thing", according to Webster's. It is used in the reflective programming community to describe the act of turning something about the implementation of a program into a data structure that the program can examine and modify.

are likely, so it is probably a good idea to double-check that a Product's sequence of MachineTools matches what its ProductType says it should be.

Applicability

Reify types when you want to make it easier to define new kinds of things at run-time. You can also use it when you want a different type system, since it lets you define your own type-checking algorithm.

Structure



Participants

- Instance - has a type

- Type

Collaborations

The Reified Type pattern divides a system into two worlds, the world of instances and the world of types. Most collaborations are between instances, or between types. These collaborations depend on the application, and so cannot be described by the Reified Type pattern.

It is common for an Instance to collaborate with its Type, because a Type holds information that is common to its Instances. An Instance usually just forwards some operations to its Type. It is much less common for Types to collaborate with Instances. A Type will often be responsible for creating its Instances, and so rarely it will keep track of the Instances that it has created.

Consequences

Reified Type makes systems more flexible at the cost of eliminating the protection provided by type checking. It makes it possible to extend a system at run-time. Unfortunately, it also makes a design more abstract, and so harder to understand, and makes a system less efficient. Debugging can be harder, because behavior is not specified in one class but in an interconnection of objects.

Implementation

Reified Type is easy to implement. Each hierarchy of Instance classes will have its own Type class. Sometimes there is a hierarchy of Type classes, but usually not. Each Instance has a Type, and an Instance delegates some subset of its behavior to its Type.

Objects have both behavior and state, and so Types must specify both the behavior and state of their instances. A common way to represent behavior is with a set of Strategies. Since a major motivation of using Reified Type is to eliminate the need to make subclasses, and since instances of new types often need new state, the Instance shouldn't store state directly in instance variables. Instead, it should have a collection of objects to store the state. In Smalltalk, this collection is usually a Dictionary. In C++, it is probably an array. The mapping from names to array offsets is done by the Type, since it is the same for all instances of the type.

The hard part of Reified Type happens after it has been used. There is an almost irresistable urge to make the Types more composable, and to build tools that let non-programmers specify new Types. These tools can get quite complex, and the structure of the Types can get quite complex. Avoid complexity unless it brings a big payoff.

Known Uses

Specializations and uses of Reified Type have already appeared in the patterns literature. Coad described an "aircraft description" object in his "Item Description" pattern, which is like Reified Type except that he only emphasized the fact that a Type holds values that all its Instances have in common[Coad92]. Hay uses Reified Type in many of his data modeling patterns, and discusses it as a modeling principle, but doesn't call it a separate pattern. He uses it to define types for activities, products, assets (a supertype of product), incidents, accounts, tests, documents, and sections of a Material Safety Data Sheet[Hay96]. Fowler talks about the separate Type and Instance worlds, and calls these the "knowledge level" and the "operational level". He uses Reified Type to define types for organizational units, accountability relationships, parties involved in relationships, contracts, the terms for contracts, and measurements, as well as many of the things that Hay discussed[Fowler97].

Reified Type is present in most reflective systems. One of the most well-known is Smalltalk, in which the class/instance separation is well known. Programmers can manipulate classes directly, adding methods, changing the class hierarchy, and creating new classes. By far the most common use of a class is to make instances, but the other uses are part of the culture and often discussed, even if not often used.

Related Patterns

Any system with a Reified Type is well on its way to having a Reflective Architecture [Buschmann96]. Often a Type holds Strategies for its instances. This is a good way to define behavior in a type.

Bibliography

[Buschmann96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns* Wiley and Sons Ltd., 1996

[Coad92] Peter Coad. Object-oriented Patterns. *Communications of the ACM,* 35(9):152-159, September 1992.

[Fowler97] Martin Fowler. *Object Blueprints: Patterns in Systems AnalysisI,* Addison-Wesley, 1997.

[Hay96] David Hay. *Data Modeling Patterns*, Dorsett House Publishing, 1996.